Extra Topics

Base Python

Try / Except - Robustness

Errors and warnings are very common while developing code, and an important part of the learning process. In some cases, they can also be useful in designing an algorithm. For example, suppose we have a stream of user entered data that is supposed to contain the user's age in years. You might expect to get a few errors or nonsense entries.

In [1]:
```
user_ages=["34", "27", "54", "19", "giraffe", "15", "83", "61", "43", "91", "sixteen"]
```

It would be useful to convert these values to a numeric type to get the average age of our users, but we want to build something that can set non-numeric values aside. We can attempt to convert to numeric and give Python instructions for errors with a `try` - `except` statement:

In [2]:
```
ages = []
problems = []

for age in user_ages:
    try:
        a = int(age)
        ages.append(a)
    except:
        problems.append(age)

print(ages)
print(problems)
```

```
[34, 27, 54, 19, 15, 83, 61, 43, 91]
['giraffe', 'sixteen']
```

User-defined Functions

While Python (and its available packages) provide a wide variety of functions, sometimes it's useful to create your own. Python's syntax for defining a function is as follows:

```
def <function_name> ( <arguments> ):
    <code depending on arguments>
    return <value>
```

The `mean` function below returns the mean of a list of numbers. (Base Python does not include a function for the mean.)

In [3]:
```
def mean(number_list):
```

```
    s = sum(number_list)
    n = len(number_list)
    m = s/n
    return m

numbers=list(range(1, 51))
print(mean(numbers))
```

```
25.5
```

Pandas

Merging and Concatenating Datasets

Concatenating Datasets

In [4]:

```
import pandas as pd
import numpy as np

a0 = pd.DataFrame({"StudentID" : [1,2],
                   "GPA_change" : np.random.normal(0,1,2)}) #np.random.normal(0,1,2) pulls 2 random
a0
```

Out[4]:

	StudentID	GPA_change
0	1	0.466944
1	2	-2.255585

In [5]:

```
a1 = pd.DataFrame({"StudentID" : [3,4],
                   "GPA_change" : np.random.normal(0,1,2),
                   "Semester" : ["Spring", "Fall"]})
a1
```

Out[5]:

	StudentID	GPA_change	Semester
0	3	0.585135	Spring
1	4	1.628179	Fall

With the datasets above it seems clear that these DataFramea would be best combined by stacking them on top of eachother, or appending one to the other as additional rows or observations. Panda's `pd.concat` lets us concatenate a list of DataFramea into a single DataFrame.

In [6]:

```
pd.concat([a0,a1],axis=0,ignore_index=True)
```

Out[6]:

	StudentID	GPA_change	Semester
0	1	0.466944	NaN

	StudentID	GPA_change	Semester
1	2	-2.255585	NaN
2	3	0.585135	Spring
3	4	1.628179	Fall

- `axis=0` indicates that we want to add the dataframes together row-wise. What happens if we change to `axis=1`?
- Pandas thinks about dimensions as *rows* and *columns*, in that order. `axis=0` refers to rows, whereas `axis=1` refers to columns.
- `ignore_index=True` resets the DataFrame index to start at 0 and run to 3. Otherwise our row index would be 0 1 0 1, from the indices of the original two DataFrames.

Merging

When joining column-wise, we usually can't just concatenate our DataFrames together, instead we use certain key variables to make sure the same observations end up in the same row.

In [7]:
```
b0 = pd.DataFrame({"name" : ["Marcos", "Crystal"],
                   "year" : [1993,1996]})
b1 = pd.DataFrame({"name" : ["Crystal", "Marcos"],
                   "proj_num" : [6,3]})

pd.concat([b0,b1], axis=1)
```

Out[7]:

	name	year	name	proj_num
0	Marcos	1993	Crystal	6
1	Crystal	1996	Marcos	3

We'll use the `pd.merge` function to merge datasets on key column variables.

In [8]:
```
pd.merge(b0,b1)
```

Out[8]:

	name	year	proj_num
0	Marcos	1993	3
1	Crystal	1996	6

`pd.merge` automatically uses all column names that appear in **both** datasets as keys. We can also specify key variables:

In [9]:
```
#pd.merge(b0,b1, on = "name")
```

```
pd.merge(b0,b1, left_on = "name", right_on = "name")
```

Out[9]:

	name	year	proj_num
0	Marcos	1993	3
1	Crystal	1996	6

Pandas also includes a DataFrame method version of `merge`:

In [10]:
```
b0.merge(b1)
```

Out[10]:

	name	year	proj_num
0	Marcos	1993	3
1	Crystal	1996	6

Note that there is also a `join` method that focuses on joining using the Pandas indices for the objects in question. It can be useful, but `merge` is usually more versatile.

In [11]:
```
b0.join(b1.set_index("name"), on="name")
```

Out[11]:

	name	year	proj_num
0	Marcos	1993	3
1	Crystal	1996	6

In the examples above, the two DataFrames share the same "name" key values. However, when the values don't completely match, we can use `how` to choose which values get kept and which values get dropped.

In [12]:
```
c0 = pd.DataFrame({"name" : ["Marcos","Crystal","Devin","Lilly"],
                   "year" : [1993,1996,1985,2001]})
c1 = pd.DataFrame({"name" : ["Marcos","Crystal","Devin","Tamera"],
                   "project_num" : [6,3,9,8]})
```

In [13]:
```
pd.merge(c0,c1, how="inner") #default merge type - keeps only observations that appear in both Data
```

Out[13]:

	name	year	project_num
0	Marcos	1993	6
1	Crystal	1996	3
2	Devin	1985	9

In [14]:

```
pd.merge(c0,c1, how="left") #The "left" join keeps only the observations in the first named DataFra
```

Out[14]:

	name	year	project_num
0	Marcos	1993	6.0
1	Crystal	1996	3.0
2	Devin	1985	9.0
3	Lilly	2001	NaN

In [15]:

```
pd.merge(c0,c1, how="right") #The "right" join keeps only the observations in the second named Data
```

Out[15]:

	name	year	project_num
0	Marcos	1993.0	6
1	Crystal	1996.0	3
2	Devin	1985.0	9
3	Tamera	NaN	8

In [16]:

```
pd.merge(c0,c1, how="outer") #The "full" or "outer" join keeps all observations
```

Out[16]:

	name	year	project_num
0	Marcos	1993.0	6.0
1	Crystal	1996.0	3.0
2	Devin	1985.0	9.0
3	Lilly	2001.0	NaN
4	Tamera	NaN	8.0

The "cross" option joins every key value to every other key value (a Cartesian product): every possible pair of names appears.

In [17]:

```
pd.merge(c0,c1, how="cross").head(10)
```

Out[17]:

	name_x	year	name_y	project_num
0	Marcos	1993	Marcos	6
1	Marcos	1993	Crystal	3
2	Marcos	1993	Devin	9
3	Marcos	1993	Tamera	8

	name_x	year	name_y	project_num
4	Crystal	1996	Marcos	6
5	Crystal	1996	Crystal	3
6	Crystal	1996	Devin	9
7	Crystal	1996	Tamera	8
8	Devin	1985	Marcos	6
9	Devin	1985	Crystal	3

Reshaping Data

Reshaping a DataFrame can have lots of benefits across data cleanup, analysis, and communication. Here are three different ways to structure the same data.

In [18]:

```
raw = pd.DataFrame({"City" : ["Raleigh","Durham","Chapel Hill"],
                    "2000" : np.random.normal(0,1,3).round(2),
                    "2001" : np.random.normal(0,1,3).round(2),
                    "2002" : np.random.normal(0,1,3).round(2),
                    "2003" : np.random.normal(0,1,3).round(2)})
raw
```

Out[18]:

	City	2000	2001	2002	2003
0	Raleigh	-0.29	0.04	0.57	-0.22
1	Durham	0.85	2.31	0.88	-0.01
2	Chapel Hill	0.70	0.02	-0.58	0.03

In [19]:

```
longer = raw.melt(id_vars = ["City"], value_vars = ["2000","2001","2002","2003"],
        var_name = "Year")
longer
```

Out[19]:

	City	Year	value
0	Raleigh	2000	-0.29
1	Durham	2000	0.85
2	Chapel Hill	2000	0.70
3	Raleigh	2001	0.04
4	Durham	2001	2.31
5	Chapel Hill	2001	0.02
6	Raleigh	2002	0.57
7	Durham	2002	0.88

	City	Year	value
8	Chapel Hill	2002	-0.58
9	Raleigh	2003	-0.22
10	Durham	2003	-0.01
11	Chapel Hill	2003	0.03

In [20]:
```
wide_again = longer.pivot(index = "Year", columns = "City", values = "value")  #reset_index moves Y
wide_again = wide_again.reset_index(col_level=1).rename_axis(columns={"City":None})  #cleanup to re
```

In [21]:
```
wide_again
```

Out[21]:

	Year	Chapel Hill	Durham	Raleigh
0	2000	0.70	0.85	-0.29
1	2001	0.02	2.31	0.04
2	2002	-0.58	0.88	0.57
3	2003	0.03	-0.01	-0.22

"Big Data" and iteration in pandas

Reminder: Download CountyHealthData_2014-2015.csv

Pandas can read csvs in smaller chunks to help deal with files that are too large to be read into RAM.

In the code below, setting `chunksize` and `iterator=True` generates a flow of 1000 row chunks out of the main dataset. This isn't really necessary in our 6109 row dataset, but might be critical to working with a 61 million row dataset.

In [22]:
```
#Create an empty list for storing chunks.
chunk_list = []

#Read in 1000 rows at a time and store only NC rows as separate chunks in chunk_list.
for chunk in pd.read_csv("CountyHealthData_2014-2015.csv", chunksize=1000, iterator=True):
    nc_rows = chunk[chunk["State"]=="NC"]
    chunk_list.append(nc_rows)

#Combine NC chunks into single data frame and view top rows.
nc_df = pd.concat(chunk_list, ignore_index=True)
```

```
nc_df.head(3)
```

Out[22]:

	State	Region	Division	County	FIPS	GEOID	SMS Region	Year	Premature death	Poor or fair health	...	Drug poisoning deaths	U
0	NC	South	South Atlantic	Alamance County	37001	37001	Region 20	1/1/2014	7123.0	0.192	...	10.48	
1	NC	South	South Atlantic	Alamance County	37001	37001	Region 20	1/1/2015	7291.0	0.192	...	12.38	
2	NC	South	South Atlantic	Alexander County	37003	37003	Region 20	1/1/2014	7974.0	0.178	...	22.74	

3 rows × 64 columns

Alternative: The `csv` package

Python also comes packaged with package for reading Comma Separated Values (csv) files. This can sometimes be easier to work with if you don't need the extra functionality of `pandas` or would prefer base Python objects to work wtih!

In [23]:
```
import csv
```

This package provides two major ways to read csv files:

- `csv.reader`: reads the csv into a list of lists where each row is represented by a list within a master list object.
- `csv.DictReader`: reads the csv into a list of **dicts** where each row is a dictionary with keys derived from the first row of the dataset.

The syntax for each command is similar:

In [24]:
```
list_of_lists = []
with open("CountyHealthData_2014-2015.csv","r") as csvfile:
    reader=csv.reader(csvfile)
    for row in reader:
        list_of_lists.append(row)
```

In [25]:
```
list_of_dicts = []
with open("CountyHealthData_2014-2015.csv","r") as csvfile:
    reader=csv.DictReader(csvfile)
    for row in reader:
        list_of_dicts.append(row)
```

Notice that each process reads the csv in row by row - this can be easily adapted with an `if` condition to filter out specific rows from a dataset that might be too large to open all at once.

Let's take a look at the differences between the output from each of these processes.

In [26]:

```
print(list_of_lists[1])
```

```
['AK', 'West', 'Pacific', 'Aleutians West Census Area', '2016', '02016', 'Insuff Data', '1/1/2014',
'', '0.122', '2.1', '2.1', '', '0.267', '0.3', '7.002', '0.234', '0.896', '0.266', '', '290.7', '2
1.1', '0.355', '91', '50', '99', '', '', '', '', '0.466', '0.091', '0.087', '', '0.289', '', '322.0
6', '', '', '0.03', '0.221', '0.272', '0', '5547', '0.078', '1', '0.067', '', '181', '', '', '0.1
7', '0.075', '', '', '0.374', '0.25', '3791', '0.185', '216', '69192', '0.127', '', '0.287']
```

In [27]:

```
print(list_of_dicts[1])
```

```
{'State': 'AK', 'Region': 'West', 'Division': 'Pacific', 'County': 'Aleutians West Census Area', 'F
IPS': '2016', 'GEOID': '02016', 'SMS Region': 'Insuff Data', 'Year': '1/1/2015', 'Premature death':
'', 'Poor or fair health': '0.122', 'Poor physical health days': '2.1', 'Poor mental health days':
'2.1', 'Low birthweight': '0.04', 'Adult smoking': '0.267', 'Adult obesity': '0.329', 'Food environ
ment index': '6.6', 'Physical inactivity': '0.22', 'Access to exercise opportunities': '0.896', 'Ex
cessive drinking': '0.266', 'Alcohol-impaired driving deaths': '', 'Sexually transmitted infection
s': '288.4', 'Teen births': '21.6', 'Uninsured': '0.293', 'Primary care physicians': '36', 'Dentist
s': '73', 'Mental health providers': '163', 'Preventable hospital stays': '', 'Diabetic screening':
'', 'Mammography screening': '', 'High school graduation': '', 'Some college': '0.474', 'Unemployme
nt': '0.088', 'Children in poverty': '0.076', 'Income inequality': '3.907', 'Children in single-par
ent households': '0.289', 'Social associations': '9.014', 'Violent crime': '317.35', 'Injury death
s': '47.2', 'Air pollution - particulate matter': '', 'Drinking water violations': '0.026', 'Severe
housing problems': '0.207', 'Driving alone to work': '0.347', 'Long commute - driving alone': '0',
'2011 population estimate': '5511', 'Population that is not proficient in English': '0.08', 'Popula
tion living in a rural area': '1', 'Diabetes': '0.065', 'HIV prevalence rate': '', 'Premature age-a
djusted mortality': '173.7', 'Infant mortality': '', 'Child mortality': '', 'Food insecurity': '0.1
73', 'Limited access to healthy foods': '0.075', 'Motor vehicle crash deaths': '', 'Drug poisoning
deaths': '', 'Uninsured adults': '0.314', 'Uninsured children': '0.176', 'Health care costs': '483
7', 'Could not see doctor due to cost': '0.185', 'Other primary care providers': '254', 'Median hou
sehold income': '74088', 'Children eligible for free lunch': '0.133', 'Homicide rate': '', 'Inadequ
ate social support': ''}
```

Read more about the `csv` package here: https://docs.python.org/3/library/csv.html

Learn more

- `pandas` provides a quick introduction here
- Python Data Science Handbook provides more detail and integration with other software.
- A full list of attributes and methods available for DataFrames is available here.